Multi-core column-store parallelization under concurrent workload

Mrunal Gawade
CWI, Amsterdam
mrunal.gawade@cwi.nl

Martin Kersten
CWI, Amsterdam
martin.kersten@cwi.nl

Alkis Simitsis
HP Labs, Palo Alto, USA
alkis@hpe.com

ABSTRACT

Columnar database systems, designed for an optimal OLAP workload performance, strive for maximum multi-core utilization under concurrent query executions. However, multi-core parallel plan generated for isolated execution leads to suboptimal performance during concurrent query execution.

In this paper, we analyze the concurrent workload resource contention effects on multi-core plans using three intra-query parallelization techniques, *static*, *adaptive*, and *cost* model parallelization. We focus on a plan level comparison of selected TPC-H queries, using in-memory multi-core columnar systems. Excessive partitions in statically parallelized plans result into heavy L3 cache misses leading to memory contention, degrading query performance severely. Overall, adaptive plans show more robustness, less scheduling overheads, and an average 50% execution time improvement compared to statically parallelized plans, and cost model based plans.

1. INTRODUCTION

The ubiquitous presence of multi-core CPUs calls for an analysis of their optimal utilization by database systems, under OLAP workloads [28, 25]. Most systems use either intra-query or inter-query parallelization to maximize multi-core utilization. Multi-core utilization represents the fraction of actual CPU cores used versus the available cores during query processing. Inter-query parallelization involves executing individual queries on each core, as used by e.g. Postgres. Intra-query parallelization involves parallelization of a query plan using the *exchange* operator, to execute on the available cores, as introduced by the Volcano system [16], and used in most commercial systems. An issue ignored in most cases is that the performance of an individual query is strongly affected by the presence of a concurrent OLAP workload, which leads to resource contention, as queries compete for shared resources such as CPU, memory, and IO bandwidth. Higher resource contention leads to extended query execution times, thereby increasing the multi-core utilization, and in turn decreasing the overall query throughput [4]. A simple solution that could be deployed is to limit the degree of parallelism of plans.

As run time resource contention is difficult to model, static and cost model based approaches cannot consider it during plan generation. Hence, quantifying the effect of a concurrent workload on an individual query's performance is difficult [31, 19]. One of the fundamental approaches is to use workload variation models to analyze their resource contention effect on a sequential query performance [11]. The resource contention problem becomes even trickier to handle during parallel plan execution, as the contention could negate the gains due to parallelism, and make identification of the degree of parallelism difficult. To gain better insights we categorize different types of workloads in a broad manner based on *inter-query* or *intra-query* parallelization modes, and analyze how the resource contention affects an individual parallelized query's performance.

We evaluate three types of intra-query parallel plan generation techniques, *static* [12], *adaptive* [14], and *cost model* [3], under concurrent OLAP workloads, using in-memory multi-core columnar database systems. Static parallelization involves row-id based range partitioning, without accounting for the resource contention. Adaptive parallelization is a new feedback based plan generation technique that performs incremental query parallelization, since many workloads use template based repeated queries. Repeated query invocations introduce more partitions in an already parallelized plan, until a plan with minimum execution time is identified. When adaptive parallel plan generation happens in the presence of a concurrent workload, it reflects the impact of resource contention. We compare both these techniques with a cost model based intra-query parallel plan generation technique.

Our main contributions are as follows.

- We evaluate the performance of parallelized plans, under different types of in-memory concurrent workloads. More partitions in *static plans* result into heavy L3 cache misses leading to memory contention. *Adaptive plans* with less partitions show up to 50% better performance. *Cost model plans* with admission control results into severe degraded performance.
- We categorize workloads based on their average CPU core idleness (which reflects their multi-core utilization), when the concurrent workload server executes in either inter-query or intra-query parallellization mode.
- We analyze the robustness of parallelized plans under

DaMoN'16, June 26-July 01 2016, San Francisco, CA, USA

 ISBN 978-1-4503-4319-0/16/06...$15.00

DOI: http://dx.doi.org/10.1145/2933349.2933350

concurrent workloads, where the select operator dominant plans exhibit more robustness than the join operator dominant plans. Overall adaptive plans show more robustness than static plans. On other hand, inter-query parallelization as used by Postgres shows much degraded performance than the column stores, but relatively more robust behavior overall.

- We highlight the influence of the operating system's default thread scheduling policy on the degree of parallelism of parallelized plans.

The paper is structured as follows. In Section 2 we provide a brief background of the static, cost model, and adaptive parallelization techniques. In Section 3 we describe the set-up for the concurrent workload to generate resource contention. We provide a detailed experimental analysis to understand the effect of resource contention in Section 4. Related work is described in Section 5. In Section 6 we summarize the lessons learned.

2. PARALLELIZATION TECHNIQUES

We use two exchange operator based columnar systems, MonetDB, an open-source operator-at-a-time execution system and Vectorwise, a commercial analytic system with pipelined vectorized execution. We implemented the static and adaptive parallelization techniques in MonetDB, as its source code was available to us. Plans are represented using an *abstract intermediate language* called MonetDB Assembly Language (MAL) [6], with operator's implementation in "C". Vectorwise on the other hand uses a cost model based parallel plan generation. Plans use typical physical algebra operator representations.

2.1 Static parallelization

Static parallelization **(SP)** already exists, as the *default* parallelization technique in MonetDB. It is done in two steps. First, the *largest* table in the input serial plan is partitioned such that the number of partitions is equal to the number of cores [12]. Next, the operators in the data flow dependent path of the partitioned base operator's data are parallelized such that the data flow pattern is maintained.

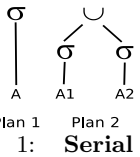

Figure 1: **Serial and Parallel plan.**

In Figure 1, Plan 1 shows a simple serial plan with the selection on A. Plan 2 shows the parallel plan derived by row-id based equi-range partitioning of A, with two new select operators. An *exchange union operator* (U) combines the parallelized operator's result. Based on the query complexity, the plans could have complex dependency patterns, and multiple exchange union operators. Aggregation is postponed as much as possible. Static parallelization is simple and fast, but could be less accurate than the cost model based parallelization.

2.2 Cost model based parallelization

The cost model based plan generation in Vectorwise involves predicting the cost of execution of a plan using an operator's input type and estimates about its cardinalities, to choose a presumably optimal plan. The data size is one of the biggest deciding factors, but memory hierarchy and access pattern, processor characteristics, and concurrent query execution affect the overall prediction considerably. Parallel plan is generated from an optimal serial plan using the exchange operator based partitioning scheme. A combination of branch and prune and dynamic programming techniques are used in the search strategy for the parallel plan.

Both static and cost model based techniques are suboptimal since the run-time resource variation can not be accounted for. Next we describe the technique that takes into account the resource contention during parallel plan generation.

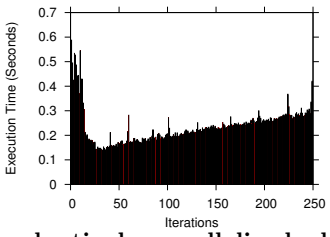

Figure 2: **An adaptively parallelized plan execution sequence for TPC-H Q14.**

2.3 Adaptive parallelization

Adaptive parallelization **(AP)** [14] is a new parallelization technique developed in MonetDB, inspired from the observation that most systems use a relatively small number of parameterized query templates repeatedly. For a more detailed description of AP, we refer the interested reader in [14]. Here we provide the core idea behind AP.

AP uses execution feedback to incrementally parallelize a query plan with each successive query invocation. A parallel plan P1 is generated from a previous plan P0, by parallelizing the most expensive operator from P0. The AP infrastructure stores previously executed plans along with the profiled information such as the operator execution time and resource claims. Under concurrent workload the execution feedback reflects the resource contention making adaptive parallel plans more robust.

Figure 2 shows an AP plan execution sequence in action where the X axis represents the consecutive invocations (iterations) of the same query. In the current setup, for feasibility purposes, we repeatedly use the same query, though most systems re-use query templates. Each vertical bar in the graph represents the plan execution time corresponding to a particular invocation. The 0th invocation corresponds to a serial plan execution, while the 1st invocation corresponds to the plan derived by parallelizing the most expensive operator from the 0th invocation plan. With consecutive invocations more operators in consecutive plans get parallelized leading to an execution time improvement until a global minimum is reached (the 19th iteration). More parallelization afterwards leads to a performance degradation. In an ideal scenario, each successive plan provides better performance than its predecessor. In practice, the execution skew due to introduction of only two partitions in successive iterations prevents it, and prolongs the convergence.

Convergence: Depending on the query complexity the number of iterations taken by AP to converge could vary. For the ease of the experimental set-up, we use a fixed number of iterations, i.e., 250 iterations which covers all possible query convergences. The convergence algorithm and various convergence scenarios are discussed in detail in [14]. For example, the TPC-H query Q14 shows minimal execution time at the 19th iteration (see Figure 2). However, the convergence runs for other queries could vary between 50 to 100 iterations.

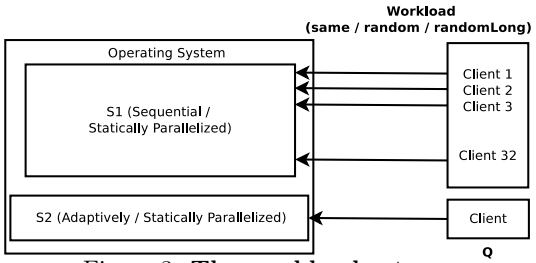

Figure 3: **The workload set-up.**

Global minimum: The speed-up of a plan is measured with respect to its serial execution. A plan is the global minimum plan if its speed-up is better than all other minimal plans observed earlier during the global minimum search. We find the global minimum execution time during multiple experiments and consider their average as the final global minimum time.

3. WORKLOAD SET-UP

In this section we describe the concurrent analytical workload set-up to generate resource contention for shared resources such as the CPU cores, to analyze its impact on a parallelized query.

3.1 Client setup

The concurrent workload consists of 32 clients connected to a MonetDB execution instance (S1 in Figure 3). As our experimental platform uses 32 CPU cores (Hyperthreaded), the number of clients are limited to 32 to ensure each CPU core has at least 1 connection. We do not aim to test the *scalability* aspect at present. The clients repeatedly fire TPC-H queries (scale factor 10) from one of the three query mix batches as shown in Table 1. The intention is not to measure *throughput*, but to keep the system always busy. The *long* queries execute in more than 1 second, where the slowest query executes in around 10 seconds. In contrast the *short* queries execute in less than 1 second.

The batch B1 consists of 32 same queries that matches the query under analysis. The batch B2 has a mix of eight short and eight long queries, and the batch B3 has ten long queries. There could be many other possible batch configurations [10], however the aim here is to show that the broad workload categorization also provides good insights into the resource contention effect on individual parallelized queries.

Table 1: **Query mix batches.**

B1	B2	B3
Same	Random	RandomLong

3.2 Server setup

We use two MonetDB execution instances S1 and S2 (see Figure 3), for the ease of experimental setup. The MonetDB execution instance **(S1)** for concurrent client connections is executed in either sequential **(Inter-query)** or statically parallelized **(Intra-query)** mode.

In sequential (SQ) mode, S1 executes a query per core such that with 32 clients, 32 queries execute on 32 cores (hyperthreaded), leading to inter-query parallelization. With this set-up we try to imitate database systems such as Postgres, which try to maximize multi-core utilization by executing a single query per core.

In statically parallelized (SP) mode, S1 does an intra-query partitioned parallel execution, such that depending on the number of row-id based range partitions, a query could get parallelized to execute on all the available cores. As a result during the concurrent workload of 32 clients, 32 queries execute in SP mode on all the available cores.

Depending on S1's execution mode and the query mix type (B1 / B2 / B3), **6 workloads** are possible as listed in **Table 3**. Our aim is to analyze the resource contention effect of these workloads on a single query's (Q) parallelized performance.

To achieve that we use a dedicated MonetDB instance **(S2)** (see Figure 3) to execute Q in AP or SP mode, in the presence of the concurrent workload executing on S1. For an AP execution a client connected to S2 repeatedly executes the same query Q for 250 iterations. Both AP and SP execution in S2 works on in-memory data without any disk IO (hot runs).

Why use separate S1 and S2 instances? Separate instances of the servers S1 and S2 allows us better instrumentation abilities for measuring the hardware events for the parallelized query (Q) under analysis. The MonetDB profiler does not use per client based connection, but has a global view of the entire execution engine. Separating S1 and S2 instances allows isolating Q's profiler statistics from the statistics of the 32 concurrent queries.

Separate execution instances however does not affect the resource contention impact from S1 on S2, as the resources such as caches, memory, and CPU cores are shared globally. MonetDB *does not* use a dedicated buffer manager, but uses a memory mapped based buffer management infrastructure. Hence, the operating system handles buffer management, thread scheduling etc. at the holistic system level.

3.3 Query set (Q)

MonetDB query plans tend to be complex due to data flow dependencies of multiple operators, represented in MonetDB Assembly Language (MAL). Hence, we identify a subset of TPC-H queries (see Table 2) to support adaptive parallelization, to analyze the concurrent workload's resource contention effect on them. The scale factor 10 is used as it provides sufficient insights about the resource contention effect. We focus on the in-memory data. The current implementation supports a single group-by attribute queries. Hence, we modify some of the existing TPC-H queries to suit this need. Since the performance of both SP and AP is analyzed using the same set of queries, the comparison is fair.

Table 2: **Query set (Q) categorization**

Simple	Q6	Q14			
Complex	Q4	Q8	Q9	Q19	Q22

3.4 Vectorwise setup

During Vectorwise experiments we use a single database instance (10GB), on which 32 concurrent clients execute queries using one of the workloads as shown in Table 3. The query Q is invoked on the same instance unlike MonetDB, so that Vectorwise plan generation resource allocation scheme could take into account the load, in terms of the number of concurrent clients. MonetDB plan generation does not take into account the run-time clients, hence having two instances S1 and S2, does not affects its plan generation.

3.5 Infinite loop workload

The workload consists of a CPU core hogging program such as a *while(1);* loop, executing on individual CPU cores on the 2 socket machine, thereby keeping them 100 % busy. The workload allows us to have a fine grained control over

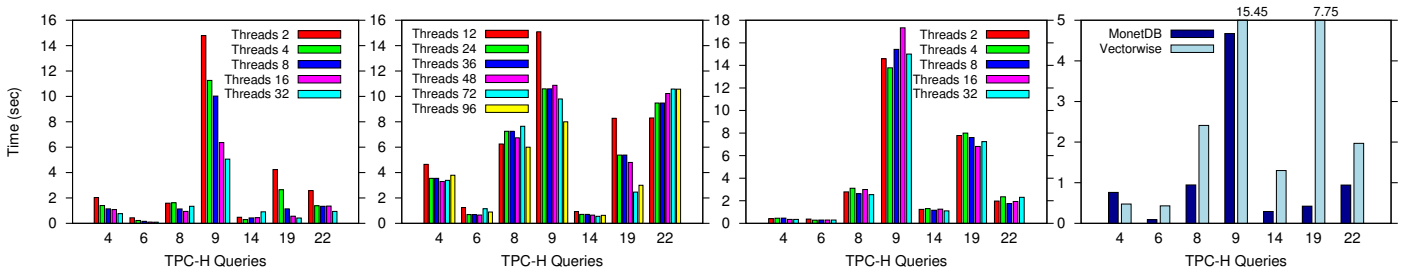

Figure 4: **Effect of degree of parallelism on query execution under concurrent workload a) MonetDB static parallelization on a 2 socket machine (10GB data). b) 4 socket machine (100GB data). c) Vectorwise cost model parallelization on a 2 socket machine (10GB data). d) Best execution time for MonetDB vs. Vectorwise using execution times from 4a and 4c.**

Table 3: % CPU core idleness for ***M***onetDB and ***V***ectorwise workloads(To be read as - ServerExecutionMode_QueryMix). Note.* - Different queries have different CPU core idleness, hence not shown.

Sequential_Same	Sequential_Random	Sequential_RandomLong	Parallel_Same	Parallel_Random	Parallel_RandomLong
15 % ***(M) *(V)***	22% ***(M)*** 27% ***(V)***	26% ***(M)*** 12.6% ***(V)***	0% ***(M) *(V)***	13%***(M)*** 27% ***(V)***	0%***(M)*** 10.6% ***(V)***

the number of busy cores at any instant. We observe the execution time of all queries on a 32 threaded statically parallelized MonetDB database instance, while the concurrent Infinite Loop workload is active. We do not enumerate cores in any specific order (like logical CPU order, socket / core/ HT order), but let the operating system use its default scheduling policy.

4. EXPERIMENTS

Our experimental setup consists of a machine with Intel Xeon E5-2650@2.00 GHz with two sockets (8 cores each), for a total of 32 threads (Hyperthreads). Cache sizes are L1=32KB, L2=256KB, L3=20MB with a memory of 256GB and Fedora 20 operating system. For some experiments we use a 4 socket machine with Intel Xeon E5-4657Lv2@ 2.40GHz, 96 threads (Hyperthreads - 24 threads / socket), L1=32KB, L2=256KB, L3=30MB, and 1TB memory with Fedora 20 operating system.

Unless otherwise mentioned all experiments use a 2 socket machine with hot execution run (no disk IO) on in-memory data (10GB). We repeat the experiments four times and report the average. We use *perf* tools to measure hardware events that reflect the resource contention impact on a subset of the queries. For the rest of the queries we use their response time as a measure to reflect the resource contention impact.

We explore the following questions in the context of different concurrent workloads.

1. How the number of partitions influence plan parallelization ?
2. Which plans perform better and exhibit more robustness?
3. Where does time go during resource contention?
4. Which is better, inter-query or intra-query parallelization?

4.1 How the number of partitions influence statically parallelized plans?

As statically parallelized plans in MonetDB are relatively simple to generate, if they are made resource contention aware, they could offer an easy solution to the plan parallelization problem. In this section we analyze if a heuristic optimizer can generate better parallel plans under concurrent workload, by tuning parameters such as the *number of partitions*.

The optimizer controls the number of partitions by controlling the number of threads. Hence, using fewer threads leads to a different plan with fewer partitions. The hypothesis is that this plan might show better concurrent workload execution performance, as it puts less pressure on the shared resources such as CPU cores and memory bandwidth due to fewer partitions. NUMA effects could also play a role. Hence, we test the hypothesis using 2 socket and 4 socket machines.

4.1.1 2 Socket NUMA

Figure 4a shows MonetDB query execution times for varying degree of parallelism, under the Parallel_Random concurrent workload, on the 2 socket NUMA machine. It nullifies our earlier hypothesis as irrespective of the number of threads in use, the minimal time occurs at 16 or 32 threads, where physical cores are 16, and 32 includes hyper-threads. Similar observations are made for other type of workloads.

This behavior could be explained by the fraction of the idle CPU cores available during the concurrent workload. When the Parallel_Random workload is used, each CPU core has an average idleness of 13% (see Table 3), which is available for the parallelized query to progress. The default operating system scheduling policy (CFS) ensures a **load balanced** fair share from all the CPU cores to the parallelized query. Hence, a 16 or 32 threaded execution (with 16 or 32 partitions) performs better than a fewer threaded execution. The queries which show better performance for 32 threads get benefited from the hyper-threads. Change of thread priorities in the Linux scheduler could alter this behavior, however we do not explore it, as we use out of the box settings.

4.1.2 4 Socket NUMA

Figure 4b shows that when the same experiment is repeated on the 4 socket NUMA machine on a 100GB dataset, the results are quite different. No explicit NUMA aware data partitioning is used as MonetDB uses memory mapped storage [13]. Execution with up to 48 threads uses the physical threads (12 threads on each socket with numactl [2] process and memory affinity), whereas 72 and 96 threaded execution also uses the hyper-threads. The behavior of each query is quite different as depending on the query complexity and NUMA access, different execution pattern is observed. For most queries around 96 threads leads to the minimal execution time, except for Q22, which shows a distinct different

behavior. Deciding exact number of partitions to give best execution is difficult[?], so partitions equal to total number of hyper-threaded cores could be a reasonable heuristic in the multi-socket machines.

4.1.3 Number of partitions and cost model plans

Figure 4c repeats the experiment for Vectorwise cost model based execution on the 2 socket machine with 10GB dataset. The first observation is irrespective of the number of threads the query execution time does not change much. We did not anticipate this behavior, because in an isolated execution setting with increasing number of threads we did see the query execution time improving with increasing threads up to 8 threads, and then staying almost constant. We do not plot this graph due to the space constraints. The scaling problems beyond 8 cores in isolated execution could be explained by [3], due to exchange operator scalability, lock synchronization issues, etc. Vectorwise's cost model based parallelization approach takes into account the load on the system in terms of the number of clients. Hence, heavy concurrent workload leads to an almost sequential execution as only a single core gets allocated for the queries under analysis. Hence, change of number of threads does not change the execution time.

Figure 4d shows the best execution time obtained using varying number of threads in MonetDB is much better than the Vectorwise time, which indicates cost model plan generation using resource allocation control might not lead to best performance under heavy concurrent workload.

We saw the influence of the number of partitions on the parallelized execution. The operating system's thread scheduling policy also has an important role to play in this setup. The micro-experiment we describe next gives more insight.

4.1.4 CPU core idleness and OS scheduling

As the concurrent parallelized queries execute on all CPU cores, controlling each CPU core's idleness is not possible, which makes the operating system's scheduling role analysis difficult. The next micro-experiment allows us a fine grained control over each CPU core's idleness, using a concurrent workload called, *Infinite Loop*, described in Section 3.5.

Figure 5 plots the execution time (Y-axis) for query 9, while varying the number of 100% busy CPU cores (X-axis) for the Infinite Loop workload. Query 9 is the longest running query in Q, hence is expected to show the largest performance variations with CPU resource share variations.

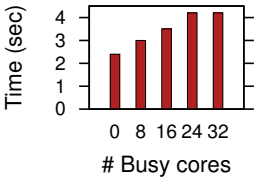

Figure 5: **Q9 with 100% busy cores when concurrent workload = Infinite Loop.**

Figure 5 shows as the number of exclusive busy cores for the Infinite Loop workload are increased, Q9 execution time increases by around 0.6 seconds. Though the cores are made 100% busy in a stepped manner, the operating system does ensure some quanta of resources on even the busy cores. Hence, the busy cores also contribute towards the parallel query execution. The idle cores contribution depends on the type of the query (CPU / memory bound). When 24 cores are busy, we observe that the operating system changes its scheduling policy and does load balancing such that now all the cores are busy. However, the cores are not 100% busy, thereby introducing some idleness on each of them.

When 32 cores are made 100% busy, since there are no more spare resources available, we do see all cores 100% busy again. However, the share of CPU resources allocated to Q9's execution does not change after the 24 busy core case. Hence, the query execution time does not change when 32 cores are made 100% busy. Some other queries do show an increase in execution time when all 32 cores are made busy. We also overload each of the CPU cores with multiple CPU core hogging programs to observe its effect on the query execution time. However, we get similar results as shown in Figure 5.

Summary: We observe on a 2 socket machine the operating system ensures a load balanced fair CPU resource share guarantee for the parallelized query execution under CPU bound concurrent workload. It ensures the best result is obtained when the number of threads equals physical cores (16) or the number of hardware contexts (32). In a 4 socket NUMA setting depending on the query complexity, different execution times are observed, largely due to remote memory accesses. However, for many queries when partitions equals either physical cores (48) or the number of hardware context (96), the best execution time is observed. Overall, hyper-threads benefit some queries. Vectorwise shows scalability issues beyond 8 threads during isolated execution, while showing much degraded performance during concurrent execution.

4.2 Static vs. adaptive vs. cost model parallelization

In this section we analyze how different plans compare from execution performance and robustness perspective. We showed that in MonetDB under different degree of parallelism, the static plans with partitions equal to the number of hardware context provide the best execution time during concurrent workload. However, these plans are not optimal, as all parallelizable operators in it use a fixed degree of parallelism. In contrast adaptive parallelized (AP) plan's operators are parallelized individually using the execution feedback, thereby allowing each parallelizable operator to have a different degree of parallelism, which helps during concurrent workload. We expect the cost model plans to show performance in between static and adaptive plans due to degree of parallelism decision based on the cost model.

Isolated execution: As a baseline reference, we use *isolated* execution comparison (see Figure 7a), where a single query executes in the system without any concurrent workload. Most queries do not show much improvement in AP compared to SP and cost model plans. As a worst case, Q19 even shows a much degraded performance in AP. It results from the presence of some non-parallelizable operators. On the other hand Q9 takes more time in MonetDB static parallelization due to multiple joins on relatively large table's attributes, as only the lineitem table is partitioned in static parallelization.

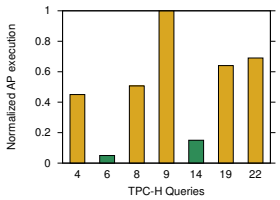

Figure 6: **Adaptive parallelized execution normalized with statically parallelized execution when concurrent workload = Parallel_Random.**

As most techniques show a comparable performance during isolated execution, we next analyze their performance during concurrent workload.

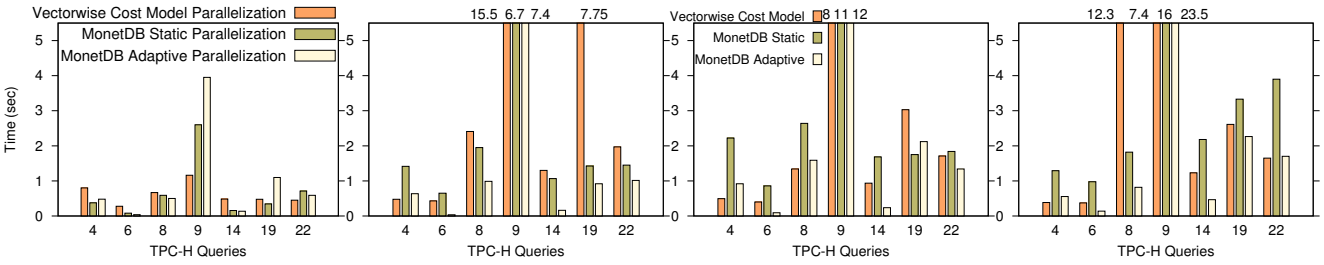

Figure 7: **a) Isolated execution. Query execution when concurrent workloads are b) Parallel_Random c) Parallel_RandomLong d) Parallel_Same.**

4.2.1 Performance

Adaptively parallelized plans perform better than the static and cost model based parallel plans, during concurrent workload due to optimal multi-core utilization.

Figure 6 gives an overview of the performance gains in AP compared to SP when the Parallel_Random concurrent workload is used. For better readability the AP execution time is normalized with respect to SP, such that, when SP execution of Q19 is 1 second, the AP execution is 0.6 seconds. While the simple queries benefit the most as is evident from Q6 and Q14 which show around 90% improvement (Green), on an average AP shows 50% improvement compared to SP for most queries. To gain better insights about individual query's performance we do an operator level analysis of some of these query plans using Figures 7a to 7d.

Query 8: Figure 7b shows that when Parallel_Random workload is used, Q8 performs better in AP than in SP. The query plan contains *join* as the most dominant operator. Plan analysis shows that the number of tuple reconstruction and join operators in the SP plan are an order of magnitude more compared to the AP plan. In many column stores tuple reconstruction operators are implemented as join operators, so they do random look-ups. Too many join and tuple reconstruction operators executing concurrently in SP plan cause costly random memory access and lead to memory bandwidth pressure. The AP plan performs better as it has a smaller number of join and tuple reconstruction operators. Vectorwise execution shows the highest time, however the performance is comparable to SP execution, except Q19.

Query 19: Figure 7b shows that when Parallel_Random workload is used, Q19 appears to perform better in AP than in SP. However, a comparison with AP from the isolated execution (see Figure 7a) shows AP execution times do not change much. AP performance looks better as SP execution is three times more expensive compared to its isolated execution.

The cause is that Q19 has the *select* operators as the dominant operators, whose parallelization during AP invocations results into the addition of new *exchange union* operators for combining their results. Based on the input selectivity the exchange union operator becomes an expensive operator after a few invocations, and gets pushed higher in the plan. However, the data flow dependencies due to a system specific non-parallelizable operator does not allow it and prevents further parallelization of Q19 plan. The SP plan does not faces this problem as it uses static partitions, which ensure the presence of *exchange union* operators much higher (i.e., later) in the plan.

Although AP performance does not change under concurrent workloads, SP shows a degraded performance as a result of resource contention due to the presence of more operators, as in Q8. We provide a detailed quantitative analysis of the resource contention effect on Q19 SP execution using a subset of hardware event measurements, in the Section 4.3.1.

In cost model plans we are not able to explain why Q19 takes the highest time. It seems though that within a database, different queries could behave differently under concurrent workload.

Summary: We observe that the AP plans has better response time than the SP, and cost model plans. The SP plans have too many operators working on fixed sized partitions, which creates scheduling and resource contention overhead under concurrent workload. Since in AP the old plan is mutated into a new plan by partitioning the most expensive operator's data, only a few operators get parallelized, where the generated partitions are dynamically sized. Some AP plans could however perform lower than the SP plans due to the presence of inherently non-parallelizable operators. Cost model plans show worst performance in most cases due to their almost sequential execution.

4.2.2 Robustness

Robustness is the ability of the database system to perform well under a variety of conditions including adverse run-time conditions due to data volume, data skew, and resource contention [31, 17]. Our focus is on the robust query processing during resource contention arising due to shared CPU cores. We consider a query plan to be robust if it gives minimal variations in the execution time under changing run-time conditions [18]. We analyze the query execution robustness by comparing its SP / AP isolated execution against the concurrent workload execution for Parallel_Random (CPU core idleness = 13%) and Parallel_RandomLong (CPU core idleness = 0) workload. Overall, the SP plans show more rapid degradation than the AP plans during concurrent workload.

Select operator: First we compare the AP execution of the queries where the select operators are dominant. Queries 4, 6, and 19 get minimally affected when Parallel_Random workload is used (see Figure 7b), while they slow down by around a factor of two under Parallel_RandomLong workload (see Figure 7c). Select operators involve either a point select or a range select operation on sequential data. As the Parallel_Random workload has average CPU core idleness = 13%, the select operators get sufficient CPU resources to execute, compared to the Parallel_RandomLong workload which has 0% average CPU core idleness.

Join operator: During AP execution the queries 8, 9, and 22 where the join operators are dominant, execute around 2

and 3 times slower for Parallel_Random and Parallel_RandomLong workloads respectively. The join operators are expensive compared to the select operators as they do random memory access keeping the CPU cores busy. As the average core idleness changes from 13% to 0% across the two workloads, their execution degrades due to insufficient CPU resources.

During the Parallel_Random and the Parallel_RandomLong workload the SP execution of the complex queries (4, 19, 8, 9, and 22) show a slowdown of 3.5 and 5 respectively, while the simple queries (6 and 14) slowdown 7 and 10 fold. The lack of CPU resources to execute many concurrent operators and the memory bandwidth pressure due to concurrent access, as we illustrate in the Section 4.3.1 is one of the main reasons for their rapid performance degradation. However, as the SP queries have too many select and join operators, isolating the exact reason for the degraded performance per operator level is difficult to access.

In Vectorwise cost model plans, we are not able to find any concrete relation between the isolated execution and the execution under Parallel_Random and Parallel_RandomLong workload. Since the execution is expected to be almost sequential due to the heavy load, the queries could show at least 16 times degraded performance compared to their isolated execution (16 is the number of physical cores). However, we do not observe that. Our observations of isolated query executions show that the cost model parallelized plans have varying degree of parallelism, unlike static parallelization. This gives rise to varying CPU core idleness, compared to MonetDB concurrent workloads, thereby making the performance under different workloads much less robust.

Query 14: AP execution of Q14 represents a special case for the Parallel_Random and the Parallel_RandomLong workloads. Its plan contains a mix of both the select and the join operators as the dominant operators, unlike the other queries which we analyzed earlier in this section. The join operators however work on much less data as it gets filtered by the select operators, making them overall less expensive. The number of select and join operators in the AP plan is much less than the SP plan, as a result Q14 gets minimally affected under both the Parallel_Random and Parallel_RandomLong workloads. Smaller number of operators allow Q14 to progress with less CPU resources, incurs minimal memory bandwidth pressure, while exhibiting a robust behavior across the workload changes.

Summary: We observe that in AP and SP the query execution robustness under resource contention is strongly influenced by parameters such as the number of operators, the type of operators, and the available CPU resources. As the select operators are cheaper than the join operators, plans where the select operators are dominant show more robust behavior compared to the plans with join operators. Overall, the AP plans are more robust than the SP plans. Cost model plan's are much less robust and their robustness is difficult to quantify.

Having seen the performance and the robustness comparison of the parallelization techniques, we investigate next how the resource contention affects them.

4.3 Where does time go during resource contention?

Resource contention could be broadly classified into software contention and hardware contention.

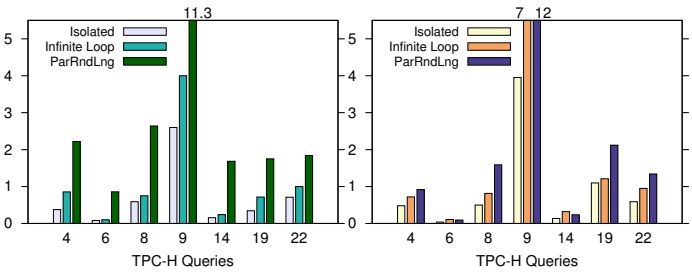

Figure 8: **a) The query execution time difference between Parallel_RandomLong (ParRndLng) and Infinite Loop workload reflects the resource contention impact on Statically b) Adaptively parallelized queries in MonetDB.**

The *software contention* arises due to the overheads in managing the shared resources such as the operating system scheduler, the lock contention manager, etc. We focus on the query scheduling overheads as the read only workload minimizes the lock contention. The *query scheduling overhead* is the time a query waits until it gets scheduled on 100% busy CPU cores.

The *hardware contention* includes data sharing conflicts resulting into (data / instruction) cache thrashing, page fault handling, TLB invalidation, context switching, etc. [9] and the CPU contention conflicts resulting into pipeline invalidation, internal units access stalls, etc. [26].

To understand where time goes during resource contention, the parallelized query execution time under concurrent workload could be dissected into the query's isolated execution, the software, and the hardware contention overhead. The difference between Parallel_RandomLong workload and the *Infinite Loop* workload execution indicates the hardware contention due to the Parallel_RandomLong workload. The difference between Infinite Loop workload and isolated execution indicates query scheduling overhead. Both workloads have 0% CPU core idleness, however differ in their work.

Software contention overhead: The query scheduling overhead indicates software contention overhead. The hardware contention is negligible as indicated by the difference between the Infinite Loop workload and the Isolated execution, as illustrated next.

As the instruction footprint of a while loop program in the *Infinite workload* is minimal, only a few CPU units such as the ones that deal with the instruction execution logic are busy, while the rest are idle. Lack of data access activity results into no cache or memory level contention as confirmed from the observations in Table 4. It shows minimal difference in query execution hardware event measures under the the Infinite Loop workload and the Isolated execution, for the SP execution of Q9.

Table 4: **Contention measure for Q9's statically parallelized execution under the Infinite Loop workload.**

	Isolated	Infinite Loop
L1 Miss %	6.6	6.5
L3 Miss %	66	58
Instructions/Cycle	.35	.41
StalledCycles/Instr	2.32	2.02

Figure 8a shows that during the SP execution the simple queries (Q6 and Q14) have minimal scheduling overhead compared to the complex queries. The SP plans have too

many operators compared to the AP plans which gets reflected in their corresponding scheduling overheads. For example, Q4 and Q19 show considerable scheduling overhead in SP execution (see Figure 8a) compared to their AP execution (see Figure 8b).

Hardware contention overhead: The hardware contention impact of the Parallel_RandomLong workload on a parallelized query execution is very high. The workload's high data access activity gives rise to heavy contention for the shared L3 cache, resulting into a large number of L3 cache misses, as could be seen in Table 5 for Q19's SP execution. It also results into heavy CPU level contention in terms of the high number of stalled instructions. We use Q19 to provide a perspective of the resource contention impact in terms of hardware performance events. For the other queries we use increased response time as a reflection of the resource contention impact.

In contrast the hardware contention for the Infinite Loop workload is negligible. We assume the query scheduling overhead for both the workloads is similar, though we expect Parallel_RandomLong workload's query scheduling overhead to be relatively more than the Infinite Loop workload's overhead, as the concurrent queries use more time quanta during their schedule. Hence, the difference between the Parallel_RandomLong and the Infinite Loop workload execution indicates the *hardware contention overhead*.

The contention overhead during AP execution (see Figure 8b) is much less compared to the SP execution (see Figure 8a) as fewer range partitioned operators execute in AP plans compared to SP plans. Fewer operators induce less scheduling overhead, and less cache thrashing.

Having established the approximate resource contention overheads in SP and AP execution in a holistic manner, we now focus on the analysis of an individual query's SP and AP execution.

4.3.1 Workload specific resource contention

We analyze the effect of the workload specific resource contention on the query execution by comparing the Parallel_RandomLong (see Figure 7c) and the Parallel_Same (see Figure 7d) workloads. Since the average idleness per CPU core is zero[1] for both the workloads, one hypothesis is, the query execution time for both should be similar. However, since that is not the case, it hints at the possibility of workload specific effects on the query execution. We explain it in the context of Q19.

Table 5: **Contention for Q19's statically Parallelized execution under different concurrent workloads.**

	Isolated	ParRandLng	ParSame
L1 Miss %	11	15	18
L3 Miss %	4	46	73
Instructions/Cycle	1.16	.21	.16
StalledCycles/Instr	.5	4.12	5.76
iCache Misses	142079	86415	79934

Query 19: The SP execution is two times slower for the Parallel_Same workload, compared to the Parallel_RandomLong workload, while AP execution for both the workloads does not show much variation. The SQL level analysis of Q19 shows the *where* clause contains a union of the results of the three sub-queries. Each of the sub-query has a range based selection predicate on the same *lineitem* table attribute with an overlapping range. The generated plan for this query takes care of maximizing sharing of the selection predicates, so that the redundant work is avoided.

[1]An exception is Q4 and Q8 in Parallel_Same workload, where average idleness is 10% and not 0%. It explains why Q4 and Q8 show much less degradation compared to other queries, where average idleness is 0%.

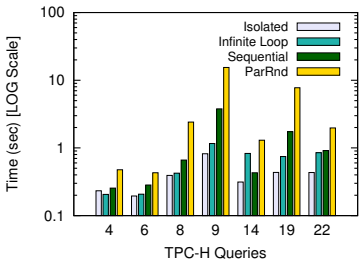

Figure 9: **Query execution under Parallel_Random (ParRnd) workload in Vectorwise degrades between 2 to 19 times compared to isolated parallel execution. Please note that the Y axis uses a log scale.**

When Parallel_Same workload is used, since the concurrent workload involves the same query, and since the query has shared predicates, it leads to access to the same base data. Since MonetDB uses memory mapped storage, these data files get shared mapping in the memory. However, storing and loading of the intermediates as they are not shared across queries generate memory bandwidth pressure. Table 5 quantifies the contention impact on Q19 and provides insight for its slow down.

It shows the percentage of the L3 cache misses is very high (73%). Very high value of L3 cache misses also indicates the pressure on the memory bandwidth. The processor pipeline is heavily stalled during the cache misses resolution, resulting in its very low utilization as seen from the Instructions per Cycle and Stalled Cycles per Instruction values. Low value of instruction cache misses compared to the isolated execution indicates the instruction cache sharing. However, any gains due to it are subdued by the dominance of the L3 cache thrashing.

In comparison, when the Parallel_RandomLong workload is used, the concurrent workload contains a mix of queries accessing different base tables. The workload also has multiple instances of Q17. Q17 contains a selection predicate on the same attribute of the *lineitem* table as in Q19. Hence, we expect some possible sharing at the memory mapped level. In [1] the authors show a matrix of TPC-H query sharing, where Q19 shares maximum data with other queries. Due to the random workload the intermediates generated are however of different sizes unlike the Parallel_Same workload thereby generating less L3 cache misses (46%), leading to a better performance. CPU utilization is also relatively more compared to the Parallel_Same workload.

4.3.2 Vectorwise resource contention

Figure 9 shows Vectorwise resource contention under different workloads. The parallelized queries under Parallel_Random workload (Yellow) are the slowest. With respect to the isolated execution the minimum slowdown is 2 times (Q4,Q6) and maximum is 19 (Q9). The performance degradation is due to resource contention and allocation scheme in Vectorwise, where the queries get resources such as CPU cores based on the existing system load. The first query gets all the available CPU cores and the subsequent queries get less cores based on a certain heuristic. In the existing scenario where the concurrent workload con-

sists of continuously executing 32 queries, the degree of parallelism for the single query under analysis is restricted to one, making its execution sequential.

To verify it we plot the sequential query execution in an isolated setting (Dark Green). The execution time difference with queries under ParRnd workload indicates the possible resource contention due to the concurrent workload on a sequential query execution. We also plot the parallelized query execution under an Infinite Loop workload executing on all CPU cores, to get an indication of the scheduling overhead for the parallelized query.

Summary: Vectorwise parallel query execution under heavy concurrent workload of random TPC-H queries shows a degradation by around 19 times compared to the isolated parallelized execution. In a similar setup, the MonetDB queries show a slow down by around four times. The observations suggest that a hard core heuristic on admission control as used by Vectorwise need not be always optimal under a heavy concurrent workload.

4.4 Inter-query vs. intra-query

Systems such as Postgres execute a single query per core. During *inter-query* parallelization, to maximize multi-core utilization multiple such queries are executed concurrently. On the other hand, most systems such as MonetDB, Vectorwise, Tableau, and SQL Server[7, 30] use *intra-query* parallelization, using the *exchange operator* [16], where a single query executes on multiple cores. We use the following setup to understand which technique performs better.

Setup: In this experiment, we compare the inter-query parallelization performance of Postgres with Vectorwise and MonetDB on 10GB data-set. Both Vectorwise and MonetDB are used in sequential execution to serve the 32 concurrent clients firing random queries (SeqRnd workload). The query Q under analysis is also executed in the sequential mode. We use Postgres version 9.4 and configure the parameters such as shared buffer size using pgtune tool recommendations. Postgres forks 32 server processes to serve the 32 concurrent clients firing continuous random queries. The client that fires query Q under analysis thus becomes the 33rd concurrent connection.

Performance: Figure 10a plots the execution performance of queries when executed in isolation vs under concurrent workload execution (SeqRnd), for the three database systems (P- Postgres, M- MonetDB, V- Vectorwise). Postgres performance in both isolated and under concurrent workload is always much lower than the corresponding MonetDB or Vectorwise performance. The much degraded performance of Postgres isolated execution is a result of its tuple-at-a-time execution engine architecture, which is not optimized for in-memory execution, unlike MonetDB and Vectorwise. An interesting observation is under concurrent workload all Postgres queries always show around two times degradation than its isolated execution. This indicates that though Postgres shows much degraded performance, still it shows a relatively robust behavior under concurrent execution.

Robustness: To test Postgres robustness behavior further, we conduct another experiment where 31 and 64 clients fire random queries under SeqRnd workload (see Figure 10b). When 31 clients fire random queries, the query Q under analysis is fired by the 32nd client. The aim of this experiment is to understand when resource of one core is available, whether the query Q gets full core for its execution and behaves similar to isolated execution, since all the 31 clients are busy with 31 cores. However, the results from Figure 10b do not indicate that. When 32 clients are active the CPU core idleness is always 0%, while when only 31 concurrent clients are active some idleness across random CPU cores is observed. Hence, the execution performance of queries when 31 clients are active is slightly better than when 32 clients are active. This indicates, possible sharing of 31 cores among 31 available clients due to lack of explicit core affinity, which prevents a dedicated single core allocation to the 32nd client for the query Q under analysis. Hyper-threading also plays its role.

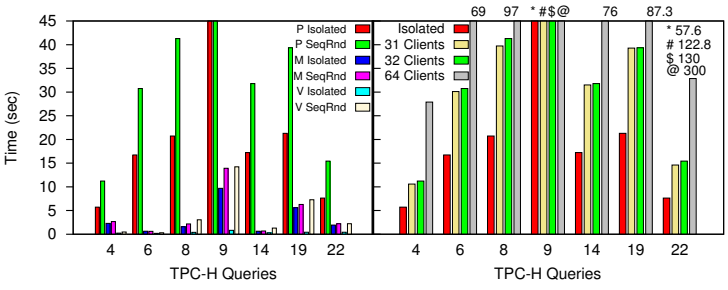

Figure 10: **a) Postgres's (P) isolated and inter-query parallelization comparison with MonetDB's (M) and Vectorwise's (V) sequential isolated execution and sequential execution under Sequential_Random (SeqRnd) workload (32 clients). b) Postgres with varying concurrent clients under SeqRnd workload.**

The execution time decreases by two and five times under 32 and 64 concurrent workload (SeqRnd), compared to isolated execution. This verifies the earlier hypothesis that Postgres execution degrades robustly with increased number of concurrent clients.

Summary: Both MonetDB and Vectorwise *inter-query* parallelized execution shows between 5 to 30 times better performance than Postgres under concurrent workload. In contrast the *intra-query* adaptive parallelized execution of MonetDB under Parallel_Random workload shows between 10 to 50 times improvement compared to Postgres's inter-query parallelized concurrent execution. However, Postgres offers a much robust execution compared to others.

5. RELATED WORK

A lot of past work deals with identifying the correct multiprogramming level (MPL) and modelling of query interactions [4, 22, 24]. A scheduling based approach is also used to model different possible query mix interactions [29].

Identifying the resource contention effect on an individual query performance has been explored in the context of sequential query execution, in the context of pipelined parallelism. Authors in [21] explore the tradeoffs of work sharing vs pipe-lined parallelism in multi-core systems in sequential query execution. In [19] authors analyze contention in chip multi-processors at different CPU cache levels.

In [5] the authors compare the behavior of three columnar systems under concurrent TPC-H and SSB queries, while scaling up the concurrent clients. It shows the response time increases linearly as the concurrent clients increase. Authors also show that throughput decreases after reaching a peak as the number of concurrent clients increase. Our work uses the response time as the performance metrics with in depth analysis from multi-core parallelization perspective. The au-

thors in [27] analyze the work / data sharing using simultaneous pipeline and global query plan techniques, however, do not discuss the parallelization aspect.

State of the art systems such as Hyper [23] use morsel driven work stealing based runtime adaptive parallelism. Here controlling the number of partitions is equivalent to controlling the size of a morsel, allowing the degree of parallelism variations of an individual query elastically. Hyper, however, is not available for a direct comparison in our setup.

In [15] the authors propose a new mechanism to minimize resource utilization and to maximize performance and predictability while deploying query plans on multi-core systems. They propose resource activitiy vectors to characterize individual database operator's behavior. A new deployment algorithm uses these vectors with dataflow information from the query plan for the optimal assignment of the relational operators to the cores. In [8] the authors introduce a new scheduling mechanism for multi-core systems where instead of CPU core oriented scheduling focus, they propose on-chip memory focused scheduling. Here threads are scheduled across cores based on their data objects usage of the on-chip memory. In [20] the authors propose Callisto, a resource management layer for parallel runtime systems. The authors illustrate how Callisto eliminates most of the scheduler-related interference between concurrent jobs, and allows jobs to claim otherwise-idle cores.

6. CONCLUSION

We compared three intra-query parallelization techniques (static, adaptive and cost model), under different in-memory multi-core columnar system, concurrent workloads.

On the TPC-H query set under evaluation under concurrent workload, the adaptively parallelized plans show more robustness and an execution time improvement of an average 50% compared to the statically parallelized plans. Static parallelization suffers due to too many partitions, which leads to severe resource contention amongst the competing threads. It leads to a large number of L3 cache misses, resulting into the memory bandwidth contention. Cost model based parallelization shows the highest time as the queries are allocated minimal CPU cores due to heavy concurrent workload. Intra-query parallelization always provides best response time than inter-query parallelization, under concurrent workload.

7. ACKNOWLEDGEMENT

We thank the MonetDB team for support. This work is supported through COMMIT grant WP-02.

8. REFERENCES

[1] co-operative scans. http://homepages.cwi.nl/ boncz/msc/2011-MichalSwitakowski.pdf.
[2] Numactl. http://linux.die.net/man/8/numactl.
[3] vectormulticore. http://homepages.cwi.nl/ boncz/msc/2010-KamilAnikijej.pdf.
[4] M. Ahmad et al. Qshuffler: Getting the query mix right. In *Proc of ICDE*, pages 1415–1417, 2008.
[5] I. Alagiannis et al. Scaling up analytical queries with column-stores. In *Proc of DBTest*, page 8, 2013.
[6] P. Boncz and M. Kersten. Mil primitives for querying a fragmented world. *The VLDB Journal*, 8(2):101–119, 1999.
[7] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, volume 5, pages 225–237, 2005.
[8] S. Boyd-Wickizer, R. Morris, M. F. Kaashoek, et al. Reinventing scheduling for multicore systems. In *HotOS*, 2009.
[9] J. Cieslewicz and K. A. Ross. Data partitioning on chip multiprocessors. In *Proc of the DAMON*, pages 25–34. ACM, 2008.
[10] U. Dayal et al. Managing operational business intelligence workloads. *Proc of SIGOPS*, pages 92–98, 2009.
[11] A. Ganapathi et al. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *Proc of ICDE*, pages 592–603. IEEE, 2009.
[12] M. Gawade and M. Kersten. Tomograph: Highlighting query parallelism in a multi-core system. In *Proc of DBTest*, page 3, 2013.
[13] M. Gawade and M. Kersten. Numa obliviousness through memory mapping. In *Proc of the DaMon*, 2015.
[14] M. Gawade and M. Kersten. Adaptive query parallelization in multi-core column stores. In *Proc of EDBT*. ACM, 2016.
[15] J. Giceva, G. Alonso, T. Roscoe, and T. Harris. Deployment of query plans on multicores. *Proceedings of the VLDB Endowment*, 8(3):233–244, 2014.
[16] G. Graefe. Volcano-an extensible and parallel query evaluation system. *KDE, IEEE Transactions on*, 6(1):120–135, 1994.
[17] G. Graefe et al. Visualizing the robustness of query execution. *arXiv preprint arXiv:0909.1772*, 2009.
[18] G. Graefe et al. Robust Query Processing. *Dagstuhl Reports*, 2(8):1–15, 2012.
[19] N. Hardavellas et al. Database servers on chip multiprocessors: Limitations and opportunities. In *CIDR*, volume 7, pages 79–87. Citeseer, 2007.
[20] T. Harris, M. Maas, and V. J. Marathe. Callisto: co-scheduling parallel runtime systems. In *Proceedings of the Ninth European Conference on Computer Systems*, page 24. ACM, 2014.
[21] R. Johnson et al. To share or not to share? In *Proc of VLDB*, pages 351–362. VLDB Endowment, 2007.
[22] S. Krompass, A. Scholz, M.-C. Albutiu, H. A. Kuno, J. L. Wiener, U. Dayal, and A. Kemper. Quality of service-enabled management of database workloads. *IEEE Data Eng. Bull.*, 31(1):20–27, 2008.
[23] V. Leis et al. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. SIGMOD, 2014.
[24] A. Mehta et al. Bi batch manager: a system for managing batch workloads on enterprise data-warehouses. In *Proc of EDBT, 640-651*, 2008.
[25] C. Mohan. Impact of recent hardware and software trends on high performance transaction processing and analytics. In *PEMCCS*, pages 85–92. Springer, 2011.
[26] H. Pirk et al. Database cracking: Fancy scan, not poor manâĂŹs sort! In *Proc of the DaMon*, 2014.
[27] I. Psaroudakis et al. Sharing data and work across concurrent analytical queries. *Proc of VLDB*, 6(9):637–648, 2013.
[28] M. Saecker et al. Big data analytics on modern hardware architectures: A technology survey. In *Business Intelligence*, pages 125–149. Springer, 2013.
[29] S. Tozer et al. Q-cop: Avoiding bad query mixes to minimize client timeouts under heavy loads. In *Proc of ICDE*, pages 397–408. IEEE, 2010.
[30] R. M. G. Wesley and P. Terlecki. Leveraging compression in the tableau data engine. In *Proc of SIGMOD*, pages 563–573, 2014.
[31] J. L. Wiener, H. Kuno, and G. Graefe. Benchmarking query execution robustness. In *Performance Evaluation and Benchmarking*, pages 153–166. Springer, 2009.